Reducing Model Complexity and Cost in the Generation of Efficient Error Detection Mechanisms

Matthew Leeke
Department of Computer Science
University of Warwick
Coventry, United Kingdom
Email: matthew.leeke@warwick.ac.uk

***Abstract*—The design and location of error detection mechanisms (EDMs) is fundamental to the design of a dependable software system. The application of machine learning algorithms to fault injection data has been shown to be an effective approach for the generation of efficient EDMs. However, the complexity of the generated models and initial cost of generation represent barriers to the adoption of the approach. Addressing these challenges directly, this paper demonstrates that genetic programming can be used as an approach to reduce the complexity of the models generated and obviate the computational cost associated with the sampling and refinement stages of EDM generation. More specifically, it is shown that (i) genetic programming can be used to project the instance space of fault injection data sets into a space more amenable to learning, (ii) machine learning algorithms can be applied to the resultant projection to permit the generation of efficient EDMs with reduced model complexity, and (iii) the cost of generating efficient EDMs can be reduced by the approach because it obviates the need for data set sampling methods and model refinement.**

***Keywords*-Complexity; Cost; Detection; Error; Genetic**

I. Introduction

The design and location of error detection mechanisms (EDMs) and error recovery mechanisms (ERMs) is a necessary activity in the development of dependable software systems, regardless of whether it is done implicitly or explicitly. EDMs are abstract components that are responsible for the detection of erroneous software states, such that these states can be addressed by ERMs before the erroneous state can lead to a violation of specification [1], [2]. It is known that error recovery is made substantially more difficult and potentially more costly if erroneous state is allowed to propagate through a software system [3].

The effectiveness of a particular EDM depends on the error detection predicate that it implements and its location in a software system [4]. Indeed, the interaction of the error detection predicate implemented and the software location can severely impact the efficiency properties of an EDM [5]. Concrete examples of EDMs include runtime assertions and parity codes, the former being the most direct realisation of the abstract component because the error detection predicate is made explicit by the implementation. The efficiency of an EDM can be measured by evaluating its accuracy and completeness properties. The accuracy of an EDM measures the rate at which it incorrectly flags erroneous software states. The completeness of an EDM measures the rate at which it correctly flags erroneous software states [4]. Hence, we can evaluate the accuracy and completeness of an implemented EDM experimentally by using software fault injection to consider its false positive rate (FPR) and true positive rate (TPR). In this context, an EDM that is entirely complete and entirely accurate is known as a perfect detector. In general it is not possible to generate or guarantee the existence of a perfect detector for a specified software location, primarily because of the read and write restrictions that are imposed on program variables, either implicitly or explicitly, during software implementation [6].

A. Learning Error Detection Predicates

It has been established that the error detection predicates for efficient EDMs can be designed by applying machine learning algorithms to the data sets generated during software fault injection [7]. Since software fault injection is a routine part of the development cycle for many dependable software systems, such data sets are often readily available at a point in development when EDMs can be incorporated. Further, the approach for generating efficient EDMs demonstrated that it was possible to produce error detection predicates for EDMs at specified software locations with near perfect accuracy and completeness. This capability has the potential to lower the reliance of software engineers on their own experience and formal specifications in the design of EDMs. The process of generating error detection predicates for EDMs using this machine learning approach comprises four stages, where these would typically be followed by the evaluation of the efficiency properties of the resultant EDMs. The stages of the process are: data collection, data preprocessing, model generation, and model refinement [8]. The EDM generation process produces first-order predicates over the set of all program variables, permitting their immediate incorporation at the prescribed locations within a software system.

Although it is not possible to generate or even guarantee the existence of a perfect detector for a specified software

location, relaxing the read and write constraints on program variables can allow for EDM detection efficiencies to be improved [6]. This benefit can be realised under various fault models and mechanisms for the implementation of software fault injection [9]. However, there are other costs associated with the generation and incorporation of efficient EDMs. More specifically, the development effort and runtime cost of incorporating a generated EDM at a specified software location increases as a function of model complexity [9]. It would therefore be desirable to reduce the model complexity of the error detection predicates generated for efficient EDMs. Further, the computational cost of the generation process depends on the data set sampling, machine learning algorithm and model refinements applied. However, the data set sampling methods, machine learning algorithms and model refinements that have been shown to be effective are adversely affected by high dimensional data sets. In response to these issues, this paper employs genetic programming to reduce the model complexity and computational cost associated with applying machine learning algorithms to high dimensional data sets in the generation of error detection predicates for efficient EDMs.

B. Contributions

This paper makes several specific contributions to the design of efficient EDMs. In particular, the research presented in this paper demonstrates that:

- Genetic programming can be used to project the instance space of fault injection data sets to a space more amenable to learning, overcoming inherent class imbalance;
- Machine learning algorithms can be applied to resultant projections to allow for the generation of efficient EDMs with reduced model complexity;
- The computational cost of generating efficient EDMs is reduced by the genetic programming approach because it obviates the need for data set sampling methods and model refinement.

The overarching contribution of this paper is to demonstrate that genetic programming can be used to reduce the model complexity and computational cost associated with applying machine learning algorithms to high dimensional data sets in the design of efficient EDMs. This is notable in the domain of software dependability, given the inherent imbalance of the data sets associated with software fault injection and the resultant challenges for the generation of error detection predicates for efficient EDMs.

II. Related Work

In this section we provide an overview of research relating to the generation of predicates for efficient EDMs. This coverage focuses on alternative mechanisms for model generation and approaches that could be adopted in order to reduce the complexity of the resultant models.

A. Generating Error Detection Predicates

The application of machine learning in the context of EDM design is appealing because it does not presume the availability of a formal specification or rely on the experience of software engineers, especially since the latter has been shown to provide inadequate detection efficiency [7], [10]. This approach has also been shown to complement metric-based approaches, since the variable incorporated by the resultant predicates are consistent with those identified by software metrics as necessary and sufficient for the detection of errors [11], [12]. In extending the approach to account for a wider variety of fault models, the experimental cost of producing the data sets that serve as input to machine learning algorithms was reduced [9]. However, the cost of generating predicates based on those data sets and the cost of incorporating predicates, particularly with regard to the associated runtime penalty, remain barriers to the adoption of the approach, particularly in a commercial context.

The runtime cost of incorporating a generated EDM at a specified software location increases as a function of model complexity [9], [13]. In the context of learning error detection predicates for efficient EDMs, the data sets collected during software fault injection will typically comprise a large number of attributes because each attribute corresponds to precisely one program variable. Moreover, the innate dependability of the software system under test and the quality of test cases considered mean that instance counts in these data sets will be necessarily large, due to the requirement for a sufficient number of examples relating to each class / outcome. In other words, it is necessary to have a viable number of instances to exemplify failure to enable the approach operate but deriving these is likely to yield data sets with many more instance of successful executions compared to failed executions.

B. Addressing Class Imbalance

The issue of class imbalance is an established problem in machine learning, not least because it makes it more difficult for algorithms to produce effective predictive models [14]. Class imbalance is commonly addressed by oversampling the minority class [15], undersampling the majority class [16], or assigning increased costs to the misclassification of the minority class [17]. It can be difficult to establish meaningful misclassification costs and greedy selection criteria do not guarantee a minimum cost model following the application of a machine learning algorithm [18], hence sampling methods are often a preferred approach when addressing the issue of class imbalance. It is typically the case that cross-validation is used to identify the extent of the sampling that should be applied to individual data sets, since this is further parameter that can not be easily specified in all situations [16].

Any undersampling process results in a loss of information because negative instances are not used in learning,

potentially impacting model quality. Despite the fact that oversampling can result in overfitting and an increase in learning time, the latter due to the necessarily larger data sets, approaches such as Synthetic Minority Oversampling Technique (SMOTE) [19] are among the most effective sampling methods. Indeed, SMOTE has been shown to be an effective technique for addressing class imbalance when learning error detection predicates for efficient EDMs.

The approach presented in this paper obviates the need for sampling methods. Instead the proposed approach uses genetic programming to create a projection of the original data set that is more amenable to the application of a machine learning algorithm, despite inherent class imbalance. Not only does this support the aim of reducing the complexity of the models generated, it also avoids the substantial computational costs associated with sampling methods that are used to support sampling-reliant machine learning algorithms.

C. Genetic Programming

Genetic programming has been shown to be an effective, and occasionally essential, precursor to the application of machine learning algorithms. It has been shown that features can be randomly constructed based on an initial set of attributes and successively evolved for fitness [20]. A genetic programming approach constructs variable length genes randomly from the set of all attributes, before randomly selecting three genes to create a chromosome to represent a possible projection of the original data set. Following the establishment of a population of chromosomes that all have a length of three, the fitness value of each chromosome is computed using a decision tree wrapper. A set of genetic operators are then applied to select the parent chromosome pairs to be used for evolution, mutation and crossover.

Where it can be shown that attribute values are distributed according to a Gaussian density function, it is possible to adopt a filter-based genetic programming approach [21]. These methods provide the same reduced model complexity sought in this paper, as well as desirable levels of accuracy where attributes are distributed according to a Gaussian density function. However, the assumption that attribute values, which correspond to the values of program variables in this context of this paper, are distributed according to a Gaussian density function makes filter-based approaches impractical in the generation of error detection predicates for efficient EDMs.

A range of constructive genetic programming approaches aim to learn multiple interacting features rather than building from a single feature [22], [23]. These approaches can provide models with improved accuracy and complexity, particularly compared to single feature approaches that are based on a greedy search, even in the context of high dimensional data sets [24]. However, as they generally rely on some form of combinatorial analysis, the computational cost of application discourages adoption in domains where high accuracy is readily achievable, as is the case in the generations of error detection predicates for efficient EDMs.

III. Models

This section details the adopted system, fault and data models, including relevant motivating assumptions.

A. System and Fault Model

The system model is identical to the modular, compositional model used in [7] and [9] to ensure consistency of approach and comparability of results. The fault injection approach introduced a single bit flip fault into the representation of a single program variable in the execution of each fault injection experiment, incorporating a single-fault assumption for the purpose of comparison with existing results relating to EDM efficiency. All variables in scope at the location under test were subject to exhaustive experimentation under this fault model. This is an established model for software fault injection, being consistent with fault models used in previous work on the application of machine learning for the generation of efficient EDMs [7], [9], [25], [26].

B. Data Model

All data collected was stored under the relational data model, where data relating to a system is modelled as a set of entities, known attributes and relationships to other entities. Data stored within the relational data model is a sample of all the data that may be generated by a system under observation. Rather than being interested in the retrieval of stored data, it is often more useful to be able to forecast behaviours of the system not previously encountered or derive knowledge about the system if it is not well understood.

IV. Genetic Programming for the Generation of Efficient Error Detection Predicates

This section provides an overview of how efficient EDMs can be generated using machine learning and how genetic programming approaches operates. The coverage of the latter includes an explanation of chromosomes, fitness functions and genetic operators.

A. Generating Error Detection Predicates

Data sets collected during software fault injection provides an indication of whether a sampled system state resulted in a failure. Hence, the generation of a predicate for an EDM from that data is a supervised learning problem. Commensurate with the data model described in Section III, data comprises of a set of n input attributes that define an n-dimensional space called the Instance Space, I. Every point in I is a potential state of the system modelled. Machine learning algorithms can then be tasked with learning an approximation, $\hat{f}$, of a target function f. This is done given a training data set, $T \subseteq I$, consisting of the N pairs $\langle x_i, f(x_i) \rangle$. In the case of learning a function from data

sets generated during software fault injection, the function is binary because a system state will either lead to a system failure or it will not. A methodology for EDM generation using machine learning on fault injection data was first described in [8]. The stages of the methodology are Data Collection, Data Preprocessing, Model Generation and Model Refinement.

B. Genetic Programming

Genetic programming operates by creating a population of candidate programs to solve a specified task [27]. An initial population is created at random, with future generation being evolved by measuring candidate fitness and selecting the most suitable candidates for reproduction. These candidate parent chromosomes generate offspring using standard genetic operators, termed crossover and mutation. As new generations are produced, assuming the absence of excessive mutation, the population converges to a solution that provides better fitness than the average of the initial population.

As the approach performs a heuristic search of the space of possible solutions, the choice of how to the measure fitness of candidates is critical. Approaches for determining fitness can be categorised as belonging to wrapper methods, embedded methods or filter methods [28]. Embedded methods measure the fitness of features based on classifier performance, which generally makes them more computationally expensive than other methods. Embedded methods share the property that they measure the fitness of features based on classifier performance but go further than wrapper methods, applying the model building metric as part of the learning process. In contrast, filter methods seek to measure the intrinsic properties of the candidates, typically using univariate statistics.

The premise of using genetic programming in this paper is to create a projection of an original fault injection data set to a lower dimensional space, such that the projection can be used to learn models with lower complexity and similar detection efficiency. By obviating the need to run sampling procedures or model refinements, this approach can also reduce the computational cost of model generation.

The genetic programming used in this paper adapts the single feature approach provided in [22]. This is done to provide an experimental lower bound for what can be achieved when using genetic programming to reduce the complexity of the models generated. The adaptation of the approach removed all restrictions on the size of the expressions that can be produced. In order to develop the genetic programming approach for this purpose, it is necessary to specify the chromosomes, fitness function and genetic operators, the latter being the mutation and crossover genetic operators in this case.

Chromosomes: Given a training data set, $D \subset R_n$, defined using a set of n attributes, A, and labelled using one of two class labels, $\{L_1, L_2\}$, the aim of the proposed approach is to learn a set of k expressions, where each expression is a function of a subset of attributes of A and constants within a continuous range. Fig. 1 shows a simple example of such an expression. The function set $F = \{+, -, \times, /\}$, noting that the division operator returns 1 when the denominator is 0.

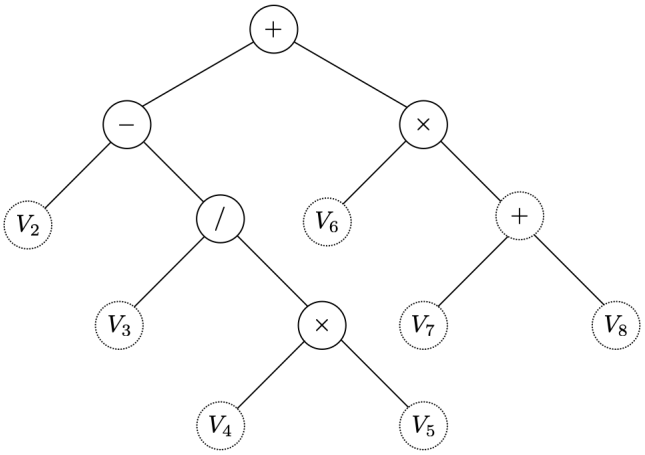

Figure 1: An example expression created from program variables in an original data set and the function set.

Fitness Function: Two filter fitness functions are explored to generate the results presented in this paper. Filter methods are less computationally expensive than wrapper-based method, since they do not use a learning algorithm in measuring chromosome fitness. Filter methods also remain independent of the learning algorithm used to generate a model from the training data set, once the genetic programming approach has selected a projection for the data. These characteristics are appealing in the context of reducing EDM model complexity and computational cost, since these aims could be realised but negated by the relocation, as opposed to obviation, of the computationally expensive aspects of model generation. This paper uses information gain and chi-squared as fitness measures for each of the expressions created. The fitness value for a chromosome is then the sum of the fitness values of each of its constituent expressions.

Crossover Operator: The mutation and crossover genetic operators are applied to candidate parent chromosomes to create a new generation. The crossover genetic operator selects a random location in each of two parent chromosomes and either exchanges the subtrees rooted at the locations or exchanges the entire expression between the two chromosomes. A crossover probability is used to decide on whether crossover should occur or whether the chosen parent chromosomes should be cloned in the next generation.

Mutation Operator: Each expression within a chromosomes are candidates for mutation. Mutation randomly selects a location within the expression and replaces the subtree rooted at that location with a randomly generated expression. The length of the randomly generated expression

inserted into the mutated expression is determined by a user-defined binomial distribution. A mutation probability is used to determine the likelihood of mutation occurring in a chromosome.

V. Experimental Approach

In this section we describe the experimental setup used to assess the efficacy of the proposed approach. Commensurate with the generation approach, the coverage addresses data collection, data preprocessing, model generation and model refinement.

A. Data Set Collection

Four target software systems were subject to experimentation: 7-Zip (7Z) compression utility [29], FlightGear (FG) flight simulator [30], MP3Gain (MG) volume normaliser [31], and ImageMagick editing suite (IM) [32]. Each target system is open source, modular and written in C/C++.

The exit points of five randomly chosen modules in each system were selected as locations. All variables in scope at the location under test were subject to exhaustive experimentation. Failures were identified through comparison with a fault-free execution, where any discrepancy in output or the completion of the test case was deemed a failure.

B. Data Set Preprocessing

Although genetic programming obviated the need to address class imbalance for the proposed approach, data preprocessing was integral for the methods that provided a baseline for comparison. As such, SMOTE was used to address class imbalance [19]. SMOTE generated synthetic examples from positive instances along the line segment joining them to one of their k-nearest positive instances. Synthetic examples were generated from each positive instance, t_{i+}. The k nearest neighbours, n_{it}'s of t_{i+} were retrieved before r of these were chosen through sampling with replacement, where r was the number of synthetic examples that each of the positive training instances contributed to the new data set. A synthetic data instance s_{ij} was generated as $\vec{s}_{ij} = \vec{t}_{i+} + q.(\vec{n}_{ij} - \vec{t}_{i+})$, where q was a random number between 0 and 1.

C. Model Generation

The decision tree induction algorithm used to generate models was C4.5 [33]. The rule induction algorithm used to generate models was Repeated Incremental Pruning to Produce Error Reduction [34]. The implementation of each algorithm was based on Weka [35].

Decision Tree Induction: Decision tree induction learns a disjunction of conjunctive rules. A decision tree comprises two types of node, these being decision nodes and leaf nodes. A decision node represents an input attribute. Each edge emanating from a decision node is labelled with one of the unique values in the domain of the attribute represented by the decision node. A leaf node represents a classification label. Each path from the root node to a leaf node can be interpreted as a set of conjunctive expressions that lead to the classification label at the leaf node. Decision tree induction is essentially a greedy search, where the algorithm repeatedly splits on the attribute that maximises the reduction in entropy at each stage of execution.

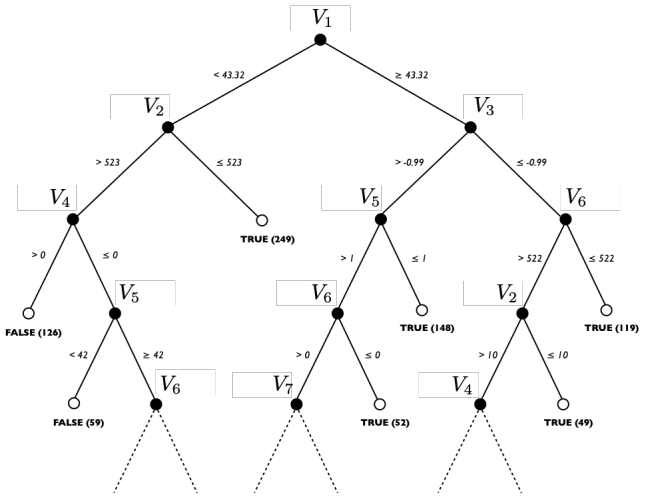

Figure 2: An example decision tree with program variables at nodes, values at edges and instance counts at leaf nodes. [7].

Fig. 2. shows an example decision tree with program variables at nodes, values at edges and instance counts at leaf nodes. In Fig. 2 the root node represents program variable V_1. Hence, each edge emanating from the root node represents a set of values for the program variable V_1. Tracing edges from V_1 results in a conjunctive expression that captures values, i.e., $(V_1 \leq 43.32) \wedge ((V_2 > 523))$. If edges are followed from the root node to a leaf node then a complete conjunctive expression will be associated with a class label and the number of instances that are captured by that conjunctive expression. For example, consistently taking the rightmost edges from the root yields the expression $(V_1 \geq 43.32) \wedge ((V_3 \leq -0.99)) \wedge (V6 \leq 522)$, where this expression accounts for 119 positive instances. The disjunction of all the conjunctive rules associated with the class label representing failure can be considered to be the predicate that has been generated for an efficient EDM.

Rule Induction: Rule Induction operates in distinct phases. Starting with with the minority class label, rule induction iteratively grows and prunes rules until there are no positive examples left or the overall error rate is greater than 0.5. A rule is grown by adding greedy conditions until the rule is accurate. This is done by attempting to incorporate every possible value of each attribute and selecting the condition providing most information gain with regard to the class label. A rule is pruned by removing any sequences of antecedents according to a pruning metric.

Further to their efficacy as machine learning algorithms for the generation of efficient EDMs, decision tree induction and rule induction are symbolic pattern learning algorithms that produce models with first order predicate representations [7], [9]. This makes them suitable for the purposes of this paper, since they can be directly implemented as runtime assertions and admit meaningful measures of model complexity. More specifically, the complexity of a model generated by decision tree induction can be given by the number of nodes in the decision tree. Similarly, the complexity of a model generated by rule induction can be given by the number of program variable usages in the rules generated by rule induction.

D. Model Refinement and Evaluation

The EDMs generated from data sets resulting from genetic programming were not subject to refinement, as the approach is based on creating a projection of the data set that is more amenable to the application of machine learning. The EDMs generated by methods that provided a baseline for comparison were subject to refinement. As in [7] and [9], the EDM refinement stage used 20 levels of sampling. These levels were uniformly distributed over [5,100] and [100,1500] for undersampling and oversampling respectively.

Following the execution of each learning algorithm to each fault injection data set, the efficiency of every generated model was computed. This included the number of true positives, the number of false negatives, the number of false positives and the number of true negatives. These values allowed for accuracy and completeness to be considered for each model generated. The area under the ROC curve (AUC) is used as a balanced measure of EDM efficiency.

Misclassification costs vary across the domains in which dependable software systems are deployed. Favourable AUC values must not achieved through the neglect of accuracy or completeness, hence FPR and TPR must also be considered for each model, since these rates directly correspond to the notions of accuracy and completeness respectively.

VI. Results

Tables I and II summarise the results presented in this paper. FPR and TPR give the mean false positive and true positive rates taken across ten cross-validation runs respectively. Var gives the AUC variance in cross-validation to confirm that genetic programming is not impairing model quality or the consistency of generation when reducing model complexity. Comp shows the average complexity of the model in cross-validation. Model complexity is characterised by the number of nodes in the decision tree in the case of decision tree induction and the number of program variable instances in the rules generated by rule induction.

A. Original Data Sets - Efficiency

Table I shows the efficiency and complexity of error detection predicates generated using original data sets according to the approach outlined in [7]. This table, with the exception of the model complexity column, can also be found in [9]. The efficiencies shown are consistent with those observed when using decision tree induction and rule induction algorithms to generate error detection predicates [7], [9]. The AUC values observed under the original data sets ranges from 0.89161 to 0.99991 for decision tree induction. The AUC values observed under the original data sets ranges from 0.88873 to 0.99780 for rule induction. A model with an AUC value in excess of 0.90 is available for every software module. Some generated EDMs were perfect with respect to accuracy, i.e., $TPR = 1$, and some perfect with respect to completeness, i.e., $FPR = 0$ but none were perfect detectors, i.e., $TRP = 1$, $FPR = 0$). It should be noted that, despite being derived using established means, these values represent state-of-the-art levels of efficiency.

B. Original Data Sets - Model Complexity

Although model complexity values are less meaningful when taken in isolation, it can be seen that rule induction has generally produced lower complexity models. Despite being the worst performing of the two machine learning algorithms with regard to efficiency, rule induction produced lower complexity models for all but two software modules. For instance, the predicates generated for EDMs in $7Z5$ under rule induction, which incorporate an average of four fewer program variables than those generated under decision tree induction. However, this is the largest difference in model complexity between the two machine learning algorithms. The mean difference in model complexity between decision tree induction and rule induction is 1.21.

Table II shows the efficiency and complexity of the models generated using the genetic programming approach to data set projection. In particular, the table shows the best results achieved under either of the information gain and chi-squared fitness functions.

C. Projected Data Sets - Efficiency

The AUC values in Table II range from 0.87476 to 0.99987 for decision tree induction and from 0.87187 to 0.99731 for rule induction. An AUC of 0.88615 or higher can be found for every software module under a combination of the machine learning algorithms, an indication that the predicates generated using projected data sets remain effective classifiers for failure inducing states. Indeed, despite some reduced levels of efficiency in several software modules, the values still represent state-of-the-art levels of EDM efficiency. The AUC of the error detection predicates for EDMs in ten software modules for decision tree induction and ten software modules for rule induction were unchanged when using projected data sets, though these were not identical sets of software modules.

Table I: The efficiency and complexity of error detection predicates generated using original data sets.

		Decision Tree Induction					Rule Induction				
Software	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**Comp**	**TPR**	**FPR**	**AUC**	**Var**	**Comp**
7Z	1	0.99849	0.00100	0.99875	6E-07	17.0	0.96456	0.00157	0.98150	7E-04	15.0
	2	0.99914	0.00009	0.99953	8E-08	19.2	0.98554	0.01241	0.98657	1E-05	18.1
	3	0.99826	0.00002	0.99912	2E-09	20.1	0.93912	0.07671	0.93120	6E-06	17.8
	4	0.95422	0.00210	0.97606	5E-04	22.6	0.94685	0.06631	0.94027	5E-05	21.2
	5	0.96010	0.00090	0.97960	5E-07	15.8	0.93022	0.06467	0.93278	1E-04	11.8
FG	1	0.79633	0.01311	0.89161	2E-05	19.2	0.94151	0.09568	0.92291	5E-04	17.7
	2	0.99982	0.00000	0.99991	2E-10	11.9	0.98244	0.00420	0.98912	5E-05	12.8
	3	0.99662	0.00111	0.99776	8E-08	09.0	0.98786	0.00033	0.99376	8E-05	08.0
	4	0.93889	0.00235	0.96827	4E-06	10.8	0.87776	0.00677	0.93550	4E-02	10.4
	5	0.94427	0.04322	0.94350	4E-04	13.2	0.92419	0.01097	0.95661	8E-04	11.3
IM	1	0.83867	0.00633	0.91617	7E-04	16.1	0.81423	0.00766	0.90329	9E-03	14.7
	2	0.86937	0.02012	0.92463	9E-05	13.0	0.82677	0.02657	0.90010	4E-03	13.0
	3	0.94789	0.00091	0.97349	1E-04	17.7	0.86754	0.00675	0.93040	5E-02	17.1
	4	0.93159	0.00459	0.96350	1E-03	16.3	0.82377	0.00950	0.90714	1E-05	17.0
	5	0.91831	0.00842	0.95495	5E-03	13.0	0.84434	0.00905	0.91765	4E-02	16.2
MG	1	1.00000	0.00990	0.99505	1E-12	15.8	0.97130	0.00001	0.98565	4E-05	14.2
	2	0.97403	0.00000	0.98702	1E-32	14.8	0.99559	0.00000	0.99780	9E-06	13.9
	3	0.99380	0.00000	0.99690	1E-32	12.0	0.90587	0.04206	0.93190	7E-07	10.2
	4	0.82290	0.01469	0.90411	3E-07	15.7	0.81036	0.00177	0.90430	2E-05	15.3
	5	0.85073	0.00349	0.92362	1E-04	11.2	0.79360	0.01614	0.88873	7E-02	09.8

Table II: The efficiency and complexity of error detection predicates generated using projected data sets.

		Decision Tree Induction					Rule Induction				
Software	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**Comp**	**TPR**	**FPR**	**AUC**	**Var**	**Comp**
7Z	1	0.99120	0.00102	0.99509	4E-07	11.9	0.96287	0.00317	0.97985	9E-04	10.9
	2	0.99914	0.00009	0.99509	8E-07	18.1	0.98554	0.01241	0.98657	1E-04	14.4
	3	0.99826	0.00002	0.99912	1E-08	19.0	0.93858	0.08095	0.92881	3E-05	14.1
	4	0.95422	0.00210	0.97606	6E-04	22.6	0.94685	0.06631	0.94027	4E-04	17.3
	5	0.95577	0.01313	0.97132	4E-07	13.7	0.92178	0.17805	0.87187	2E-04	09.9
FG	1	0.79633	0.01311	0.89161	1E-04	19.2	0.94151	0.09568	0.92292	6E-04	11.4
	2	0.99974	0.00001	0.99987	1E-08	09.4	0.98244	0.00420	0.98912	5E-05	09.2
	3	0.99662	0.00111	0.99776	2E-06	06.2	0.98770	0.02321	0.98224	7E-04	06.2
	4	0.92711	0.03768	0.94471	1E-04	10.1	0.87674	0.01389	0.93142	4E-02	05.9
	5	0.94302	0.06314	0.93994	5E-05	09.7	0.91262	0.02303	0.94480	9E-04	08.2
IM	1	0.83867	0.00633	0.91617	6E-04	14.2	0.81423	0.00766	0.90329	3E-02	11.1
	2	0.86937	0.02012	0.92463	7E-05	09.7	0.82677	0.02657	0.90010	6E-03	09.6
	3	0.94789	0.00091	0.97349	9E-05	17.7	0.85122	0.01241	0.91941	6E-02	16.9
	4	0.93159	0.00462	0.96349	2E-04	11.8	0.82185	0.03161	0.89512	2E-04	08.1
	5	0.91831	0.00855	0.95488	1E-03	12.0	0.84434	0.00905	0.91765	5E-02	10.7
MG	1	1.00000	0.00990	0.99505	8E-10	15.8	0.97130	0.00001	0.98565	1E-04	14.8
	2	0.97403	0.00002	0.98701	2E-14	09.8	0.99550	0.00088	0.99731	2E-06	09.6
	3	0.99380	0.00000	0.99690	1E-18	09.6	0.90587	0.04206	0.93191	1E-05	06.6
	4	0.82290	0.01472	0.90409	3E-06	13.2	0.81036	0.00177	0.90430	1E-04	12.2
	5	0.85019	0.10067	0.87476	3E-04	07.4	0.78924	0.01693	0.88615	8E-02	07.3

D. Projected Data Sets - Model Complexity

The results presented in Table II demonstrate that the use of the genetic programming approach impacts but does not impair the performance of the models generated beyond application. As such it can be expected that, in at least some domains, the benefits of reducing the model complexity and computational cost would justify a potential, and likely marginal, reduction in detection efficiency. The reduction in model complexity for decision tree induction ranges from 0 to 6.2, with a mean of 2.42. The reduction in model complexity for rule induction ranges from 0 to 8.9, with a mean of 3.58. The mean difference in model complexity between decision tree induction and rule induction is 2.37, a reduction that is potentially due to the rule induction appearing to generate models that are inherently less complex despite having poorer efficiencies.

The use of genetic programming reduced model complexity in all but three cases, where those cases account for just two software modules. More specifically, two models could not be made simpler by decisions tree induction and one model could not be made simpler by rule induction. It is notable that $MG1$, the case where both algorithms failed to

reduce model complexity, is a software module with near-perfect detection efficiency ($AUC = 0.99505$) based on original data sets.

The low variance of the models generated using genetic programming demonstrates that efficient EDMs can still be consistently designed using the proposed approach. Despite this, the variance values for these models are consistently higher than those of models generated using original data sets. Nonetheless, even the highest variance for a model generated using the proposed approach (0.8 for $MG5$) is indicative of a consistent and repeatable model. This consistency of model generation is most pronounced in the case of decision tree induction.

The results presented demonstrate that the projected data sets produced by the proposed approach can be used to generate predicates for efficient EDMs with reduced model complexity. In many cases this reduction in model complexity was achieved with no loss of detection efficiency. This is despite the fact that the genetic programming approach does not require the computationally expensive sampling or model refinement steps that have been intrinsic to previous EDM generation methodologies [7]. The removal of these obstacles addresses two barriers to the adoption of a software fault injection, data-driven approach for efficient EDM design. Moreover, the results presented is this paper have applied a comparatively elementary projection approach. It is possible that applying more sophisticated approaches, such as deep learning [36] or combined feature selection [37] techniques, could further improve model complexity and reduce generation cost without compromising detection efficiency.

VII. Conclusion

This paper demonstrated the efficacy of genetic programming as a means to reduce the model complexity and computational cost associated with applying machine learning algorithms to high dimensional data sets in order to generate efficient EDMs. The proposed approach obviates the need to address the class imbalance that is inherent to fault injection data, whilst yielding detection efficiencies that are commensurate with those achieved using original data sets.

The results presented pave the way for the consideration of more advanced evolutionary approaches to data set projection for EDM design. The genetic programming used in this paper focused on the creation of a projection by construction using single features of variable length. This represents a promising lower bound on what can be achieved with regard to the use of genetic programming in this domain but does not account for the use of genetic programming approaches that account for multiple features with complex interactions, this being a significant area for future research. Similarly, the fitness functions explored in this paper are limited to filter methods, leaving opportunities for the consideration of wrapper methods to improve on the efficiency of the lower complexity models learnt from projected data sets.

References

[1] J.-C. Laprie, *Dependability: Basic Concepts and Terminology*. New York, USA: Springer-Verlag, December 1992.

[2] A. Arora and S. S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," in *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*. Amsterdam, Netherlands: IEEE Computer Society, May 1998, pp. 436–443.

[3] D. Powell, E. Martins, J. Arlat, and Y. Crouzet, "Estimators for fault tolerance coverage evaluation," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 261–274, June 1995.

[4] A. Jhumka, F. Freiling, C. Fetzer, and N. Suri, "An approach to synthesise safe systems," *International Journal of Security and Networks*, vol. 1, no. 1, pp. 62–74, September 2006.

[5] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*. Budapest, Hungary: IEEE, June 2013, pp. 1–12.

[6] A. Jhumka and M. Leeke, "Issues on the design of efficient fail-safe fault tolerance," in *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering*. Mysuru, India: IEEE Computer Society, November 2009, pp. 155–164.

[7] M. Leeke, A. Jhumka, and S. S. Anand, "Towards the design of efficient error detection mechanisms for transient data errors," *The Computer Journal*, vol. 56, no. 6, pp. 674–692, June 2013.

[8] M. Leeke, S. Arif, A. Jhumka, and S. S. Anand, "A methodology for the generation of efficient error detection mechanisms," in *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2011, pp. 25–36.

[9] M. Leeke, "Simultaneous fault models for the generation and location of efficient error detection mechanisms," *The Computer Journal*, vol. https://doi.org/10.1093/comjnl/bxz022, April 2019.

[10] N. G. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall, "The use of self checks and voting in software error detection: An empirical study," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 432–443, April 1990.

[11] J. Fairbrother and M. Leeke, "On basis sets variables for efficient error detection," in *Proceedings of the 15th IEEE International Conference on Dependable, Autonomic and Secure Computing*. Florida, USA: IEEE, November 2017, pp. 399–406.

[12] A. Jhumka and M. Leeke, "Early identification of locations for dependability components in dependable software," in *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering*. IEEE, November 2011, pp. 40–49.

[13] M. Leeke, "Simultaneous fault models for the generation of efficient error detection mechanisms," in *Proceedings of the 28th IEEE International Symposium on Software Reliability Engineering*. Toulouse, France: IEEE Computer Society, 2017, pp. 112–123.

[14] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera, "A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, no. 4, pp. 463–484, July 2012.

[15] J. Van Hulse, T. M. Khoshgoftaar, and T. M. Napolitano, "An empirical comparison of repetitive undersampling techniques," in *Proceedings of the 10th IEEE International Conference on Information Reuse and Integration*. Nevada, USA: IEEE, August 2009, pp. 29–34.

[16] N. V. Chawla, D. A. Cieslak, L. O. Hall, and A. Joshi, "Automatically countering imbalance and its empirical relationship to cost," *Journal of Data Minining and Knowledge Discovery*, vol. 17, no. 2, pp. 225–252, February 2008.

[17] K. M. Ting, "An instance-weighting method to induce cost-sensitive trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 3, pp. 659–665, May 2002.

[18] M. J. Pazzani, C. Merz, P. Murphy, K. Ali, T. Hume, and C. Brunk, "Reducing misclassification costs," in *Proceedings of the 11th International Conference on Machine Learning*. New Jersey, USA: Morgan Kaufmann, July 1994, pp. 217–225.

[19] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, no. 1, pp. 321–357, May 2002.

[20] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, April 1975.

[21] K. Neshatian, M. Zhang, and M. Johnston, "Feature construction and dimension reduction using genetic programming," in *Proceedings of the 20th Australian Joint Conference on Artificial Intelligence*. Springer-Verlag, December 2007, pp. 160–170.

[22] M. G. Smith and L. Bull, "Genetic programming with a genetic algorithm for feature construction and selection," *Programming and Evolvable Machines*, vol. 6, no. 3, pp. 265–281, August 2005.

[23] L. S. Shafti and E. P. Pérez, "Constructive induction and genetic algorithms for learning concepts with complex interaction," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. Washington DC, USA: ACM, June 2005, pp. 1811–1818.

[24] B. Tran, B. Xue, and M. Zhang, "Genetic programming for multiple-feature construction on high-dimensional classification," *Pattern Recognition*, vol. 93, pp. 404–417, September 2019.

[25] M. Leeke and A. Jhumka, "An automated wrapper-based approach to the design of dependable software," in *Proceedings of the 4th International Conference on Dependability*, Nice, France, August 2011, pp. 43–50.

[26] D. Powell, "Failure model assumptions and assumption coverage," in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*. Wisconsin, USA: IEEE Computer Society, July 1992, pp. 386–395.

[27] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, February 1993.

[28] K. Krawiec, "Genetic programming-based construction of features for machine learning and knowledge discovery tasks," *Genetic Programming and Evolvable Machines*, vol. 3, no. 4, pp. 329–343, December 2002.

[29] 7-Zip, "https://www.7-zip.org/," October 2021.

[30] FlightGear, "https://www.flightgear.org/," October 2021.

[31] MP3Gain, "http://mp3gain.sourceforge.net/," October 2021.

[32] ImageMagick, "https://www.imagemagick.org," October 2021.

[33] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, October 1992.

[34] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the 12th International Conference on Machine Learning*, July 1995, pp. 115–123.

[35] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, June 2009.

[36] S. Wang, Z. Ding, and Y. Fu, "Feature selection guided auto-encoder," in *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. California, USA: AAAI Press, February 2017, pp. 2725–2731.

[37] H. Yao and L. Tian, "A genetic-algorithm-based selective principal component analysis (ga-spca) method for high-dimensional data feature extraction," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 41, no. 6, pp. 1469–1478, June 2003.